Capitolo 10

Correttezza dei programmi

In questo capitolo affrontiamo un tema classico dei metodi formali: la correttezza dei programmi scritti in linguaggi imperativi (o procedurali), come C, C++ o JAVA. L'obiettivo che ci poniamo consiste nel fornire metodologie e tecniche per dimostrare formalmente che funzioni scritte in tali linguaggi sono corrette, ovvero soddisfano le condizioni scritte durante la fase di specifica. Per fare ciò porremo in maniera chiara le ipotesi sintattiche sui programmi imperativi analizzati e ne daremo la semantica in termini di grafi di flusso e logica temporale. In tale maniera saremo pronti per fornire interessanti proprietà quali la *terminazione*, la *correttezza parziale* e la *correttezza totale* come proprietà espimibili in LTL. Infine, mostreremo come validare in maniera automatica tali proprietà, usando lo strumento NUSMV.

10.1 Sintassi dei programmi imperativi

Una premessa essenziale riguarda la complessità dei problemi di verifica delle proprietà dei programmi, in quanto tali problemi non sono decidibili in generale. In questo paragrafo descriviamo le caratteristiche dei programmi che prenderemo in considerazione, e le condizioni in cui ci poniamo sono tali da garantire la decidibilità delle verifiche proposte (ad esempio la terminazione). In particolare dobbiamo limitare la memoria a disposizione dei programmi, e garantire che venga usata in maniera coerente.

Riportiamo ora le ipotesi sui programmi che utilizzeremo. Tenendo conto che in linea di principio è possibile usare qualsiasi linguaggio imperativo, come linguaggio di programmazione per i nostri esempi utilizzeremo JAVA, con alcune limitazioni appresso elencate.

Unità: ci concentreremo su proprietà di *unità* di programma, chiamate in JAVA *metodi* o *funzioni*.

Il motivo di questa scelta risiede nel fatto che è prassi comune (cfr. ad es., [31, 34]) iniziare le attività di verifica dei programmi dalle singole unità, per procedere successivamente alla verifica di integrazione e di sistema.

Memoria finita: prenderemo in considerazione unità di programma che fanno uso di una quantità di memoria *finita* e stabilita a priori.

Il motivo di questa scelta risiede nella indecidibilità del problema della verifica per programmi con memoria illimitatata (cfr. ad es., [26, 27]). In questo capitolo siamo interessati a fornire metodi pratici e limitati nello scopo: la finitezza della memoria garantisce che il numero di stati che il programma può attraversare sia finito, e quindi implica la decidibilità della sua verifica.

Memoria non condivisa: prenderemo in considerazione unità di programma che, se usano memoria dinamica o riferimenti tramite puntatori, non denotano mai tale memoria per mezzo di più identificatori.

Infatti le metodologie proposte si riferiscono a programmi in cui esista una corrispondenza biunivoca fra *nomi* delle variabili e *celle di memoria* utilizzate per immagazzinare i loro valori. Nel momento in cui adottiamo la condivisione di memoria tramite puntatori tale corrispondenza non è più osservata, in quanto a diversi nomi corrispondono identiche celle di memoria.

Per quanto riguarda la prima ipotesi, assumeremo che la computazione dell'unità di programma si fermi in corrispondenza delle istruzioni di uscita dalla funzione, come l'ultima istruzione o istruzioni di `return` o `exit()`.

```
// File Massimo.java

public class Massimo {
    public final static int MIN = 0;
    public static void stampaVettore (int[] vett) {
        for (int i = 0; i < vett.length; i++)
            System.out.print(i + ":" + vett[i] + ", ");
        System.out.println();
    }
    public static int massimo(int[] vett) {
        // precondizione: gli elementi appartengono all'intervallo MIN..maxint
        int result = MIN;          /* 1 */
        int i = 0;                 /* 2 */
        while(i < vett.length) {   /* 3 */
            if (vett[i] > result)  /* 4 */
                result = vett[i];  /* 5 */
            i++;                   /* 6 */
        }
        return result;             /* 7 */
    }
    public static void main(String[] args) {
        int[] a = { 1, 5, 3 };
        stampaVettore(a);
        System.out.println("massimo: " + massimo(a) +
                           "\n***********");
    }
}
```

Figura 10.1 Programma JAVA per il calcolo del massimo in un vettore di interi

Un commento sulla seconda ipotesi è doveroso. L'assunzione della limitatezza della memoria pone un severo limite alla validità delle conclusioni sui programmi che siamo in grado di trarre. Ad esempio, dimostrare che un programma per l'ordinamento di un vettore è corretto per vettori di al più dieci elementi non implica affatto che tale programma sia corretto per tutti i vettori, e in particolare nemmeno per quelli di undici elementi. Va tuttavia ricordato che le tendenze contemporanee dei metodi formali nell'ingegneria del software pongono molta enfasi sulle tecniche di validazione basate su *piccoli esempi*, in quanto la pratica mostra che moltissimi errori di analisi, codifica, etc. emergono anche in situazioni limitate. Ad esempio abbiamo visto tale metodologia concretizzata nel sistema ALLOY (cfr. paragrafo 7.4.2.2), in cui è obbligatorio dichiarare la dimensione massima del dominio di oggetti di interesse. Possiamo quindi dire che dai metodi esposti in questo capitolo non ci aspettiamo di dimostrare incondizionatamente la correttezza dei programmi, ma piuttosto di evidenziare la presenza di errori che, attraverso un'indagine empirica e non formalizzata, sfuggirebbero.

Riportiamo nelle figure 10.1 e 10.2 due programmi che utilizzeremo per gli esempi. I programmi si riferiscono a due semplici problemi:

1. ricerca del massimo in un vettore di interi, e
2. ordinamento di un vettore di interi; l'ordinamento viene ottenuto mediante il noto algoritmo *a bolle* (cfr. ad es., [12]).

Per quanto riguarda le ipotesi generali formulate in precedenza, chiariamo alcuni aspetti.

- Le unità di programma che ci interessano sono, rispettivamente:
  - `int massimo(int[])`, e
  - `int[] bubbleSort(int[] vett)`.

  I programmi completi vengono forniti con lo scopo di aumentare la comprensibilità degli esempi.
- L'ipotesi sulla finitezza della memoria viene ottenuta limitando i valori delle variabili e le dimensioni delle strutture di dati. Dobbiamo quindi dichiarare in maniera precisa:
  - quali siano le dimensioni massime dei vettori, e

```
// File BubbleSort.java

public class BubbleSort {
    public static void stampaVettore (int[] vett) {
        for (int i = 0; i < vett.length; i++)
            System.out.print(i + ":" + vett[i] + ", ");
        System.out.println();
    }
    public static int[] bubbleSort(int[] vett) {
        // ordina in maniera crescente -- no side-effect
        int[] result = new int[vett.length];
        int i = 0;                              /*  1 */
        while (i < vett.length) {               /*  2 */
            result[i] = vett[i];                /*  3 */
            i++;                                /*  4 */
        }
        int temp;
        int index = vett.length - 1;            /*  5 */
        while(index >= 0) {                     /*  6 */
            int j = vett.length - 1;            /*  7 */
            while(j > 0) {                      /*  8 */
                if (result[j] < result[j-1]) { /*  9 */
                    temp = result[j];           /* 10 */
                    result[j] = result[j-1];    /* 11 */
                    result[j-1] = temp;         /* 12 */
                }
                j--;                            /* 13 */
            }
            index--;                            /* 14 */
        }
        return result;                          /* 15 */
    }
    public static void main(String[] args) {
        int[] a = { 7, 5, 3, 1, -3 };
        stampaVettore(a);
        stampaVettore(bubbleSort(a));
    }
}
```

Figura 10.2 Programma JAVA per l'ordinamento di un vettore di interi con l'algoritmo *a bolle*

  - i valori minimo e massimo per ognuna delle variabili locali e parametri formali delle unità.

  Ad esempio, con riferimento alla figura 10.1, le seguenti assunzioni sono adeguate:

  - le variabili locali `max` e `i` sono comprese fra `-10` e `10`;
  - il vettore di interi `vett` ha al più 5 elementi;
  - tali elementi sono compresi fra `0` e `10`;

- L'unità di programma `bubbleSort()` fa uso di memoria dinamica per allocare il risultato (vettore `result`), ma tale memoria non è condivisa da altri identificatori.

10.2 Semantica dei programmi imperativi

In questo paragrafo forniamo il significato formale di un'unità di programma del tipo descritto nel paragrafo 10.1. Esistono molte maniere di descrivere tale semantica; quella più comoda per i nostri scopi fa riferimento alla logica temporale lineare LTL, introdotta nel capitolo 6. Infatti le ipotesi di finitezza della memoria poste in precedenza fanno sì che un un'unità di programma possa essere efficacemente modellata mediante un sistema di transizioni.

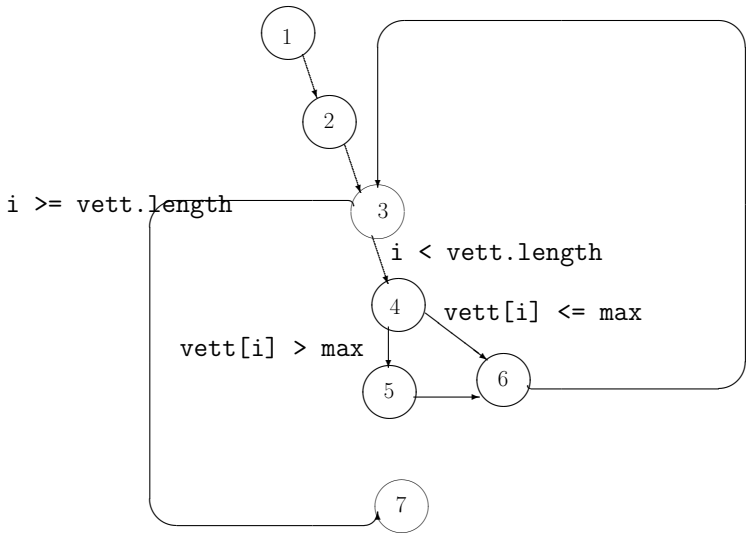

Figura 10.3 Grafo di flusso per la funzione `massimo()` di figura 10.1

10.2.1 Grafi di flusso

Per rendere più chiara la corrispondenza fra programmi e sistemi di transizione è possibile fare ricorso ai cosiddetti *grafi di flusso*, che consistono in rappresentazioni non struttutate dei programmi, molto note nell'Informatica e usate sin dagli inizi di questa disciplina.

Nei grafi di flusso le singole istruzioni diventano nodi, mentre le possibili sequenze fra istruzioni (multiple, in presenza di istruzioni condizionali o di ciclo) diventano insiemi di archi in uscita dai nodi corrispondenti. Per facilitare la leggibilità, le istruzioni e i nodi vengono etichettati con interi e gli archi vengono etichettati (quando necessario) con le condizioni di attraversamento. Ad esempio, la figura 10.3 riporta il grafo di flusso per l'unità di programma `massimo()` di figura 10.1.

Possiamo costruire una precisa corrispondenza fra un grafo di flusso per un'unità di programma `f()` e un sistema di transizioni $M = (S, R, L)$ di LTL secondo le regole indicate di seguito.

- L'insieme **LP** di lettere proposizionali di M è sufficiente a rappresentare:
  - tutti i parametri formali di `f()`;
  - tutte le variabili locali di `f()`;
  - il *contatore di programma* (o *PC*, dall'inglese *program counter*) di `f()`, ovvero una variabile speciale che memorizza l'istruzione in esecuzione.

  I valori di queste entità non sono booleani, ma è sempre possibile trovare una corrispondenza adatta allo scopo.
- I possibili stati S di M corrispondono al PC e alle variabili di `f()`.
- La relazione di transizione R di M codifica le possibili sequenze di esecuzione delle istruzioni di `f()`.
- La funzione di etichettatura L di M rappresenta i valori del PC e delle variabili di `f()` nei vari stati.

Il grafo di flusso evidenzia solamente l'evoluzione del PC, quindi dobbiamo immaginare che in M esista una "copia" dei suoi nodi e dei suoi archi per ognuno dei valori ammissibili delle variabili di `f()`.

Per completezza riportiamo alcuni esempi che chiarificano la costruzione dei grafi di flusso per i principali costrutti dei linguaggi di programmazione imperativi:

- **sequenza:** cfr. figura 10.4;
- `if`**:** cfr. figura 10.5;
- `switch`**:** cfr. figura 10.6;

```
// File Sequenza.java
public class Sequenza {
  public static void main(String[] args) {
    /* 1 */    int x = InOut.readInt(); // lettura da tastiera
    /* 2 */    System.out.println(x);
  }
}
```

Figura 10.4 Grafo di flusso per la sequenza di istruzioni

```
// File If.java
public class If {
  public static void main(String[] args) {
    int x = InOut.readInt(); // lettura da tastiera
    /* 1 */  if (x >= 0)
    /* 2 */    System.out.println("Positivo o nullo");
             else
    /* 3 */    System.out.println("Negativo");
    /* 4 */  System.out.println("Arrivederci!");
  }
}
```

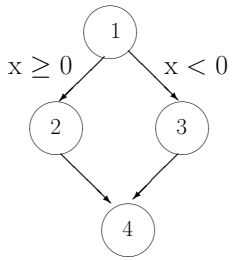

Figura 10.5 Grafo di flusso per l'istruzione `if`

- `while`: cfr. figura 10.7;
- `for`: cfr. figura 10.8;
- `do`: cfr. figura 10.9;
- `return`: cfr. figura 10.10, parziale.

Esercizio 10.1 [Soluzione a pag. 159] Costruire il grafo di flusso per l'unità di programma `bubbleSort()` di figura 10.2. ◦

Per ultimo vogliamo notare la somiglianza fra grafi di flusso e diagrammi UML degli stati e delle transizioni (cfr. paragrafo 8.1). Intuitivamente i primi possono essere visti come diagrammi UML in cui le transizioni sono volte a fare cambiare il PC del programma, modificando anche le variabili di stato, ovvero i valori delle variabili del programma.

10.3 L'uso della logica del prim'ordine per esprimere proprietà di programmi

In questo paragrafo mostreremo un'interessante applicazione della logica alla progettazione dei programmi. Useremo la logica del prim'ordine per esprimere precisamente proprietà dei programmi. Introdurremo l'argomento mostrando che la logica consente di esprimere proprietà degli stati della computazione di un programma (paragrafo 10.3.1) e successivamente illustreremo un metodo per specificare precondizioni e postcondizioni del calcolo di una funzione (paragrafo 10.3.2).

Al contrario della trattazione vista nel paragrafo 10.2, relativa solamente a programmi con memoria finita, in questo paragrafo vedremo nozioni valide per tutti i programmi.

10.3.1 Asserzioni su uno stato del programma

Il primo modo di utilizzare la logica per esprimere proprietà dei programmi è relativo all'idea di specificare proprietà di uno stato della computazione. In questo caso, utilizziamo una formula della logica del prim'ordine per rappresentare una condizione che deve essere verificata in un particolare stato della computazione del programma. Esprimeremo queste condizioni mediante commenti inseriti nel codice del programma stesso.

Ad esempio, consideriamo una funzione Java `f()`:

```
// File Switch.java
public class Switch {
  public static void main(String[] args) {
    int x = InOut.readInt(); // lettura da tastiera
    /* 0 */ switch (x) {
    /* 1 */ case 1: System.out.println("Uno");
              break;
    /* 2 */ case 2: System.out.println("Due");
              break;
    /* 3 */ case 3: System.out.println("Tre");
              break;
    /* 4 */ default: System.out.println("Errore");
    } /* 5 */
  }
}
```

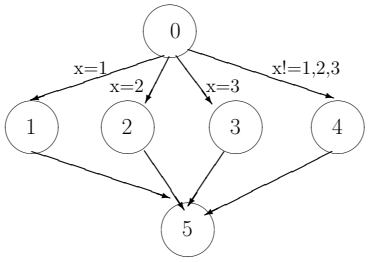

Figura 10.6 Grafo di flusso per l'istruzione `switch`

```
// File While.java
public class While {
  public static void main(String[] args) {
    int x = InOut.readInt(); // lettura da tastiera
    /* 1 */ while (x > 0) {
    /* 2 */   System.out.println(x);
    /* 3 */   x--;
    } /* 4 */
  }
}
```

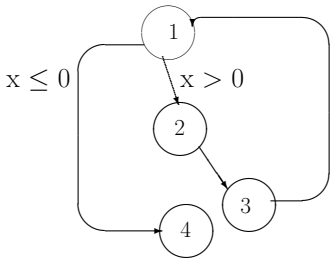

Figura 10.7 Grafo di flusso per l'istruzione `while`

```
// File For.java
public class For {
    public static void main(String[] args) {
        for (/* 0 */
             int x = InOut.readInt();
             /* 1 */
             x > 0;
             /* 3 */ x--
             )
            /* 2 */ System.out.println(x);
    } /* 4 */
}
```

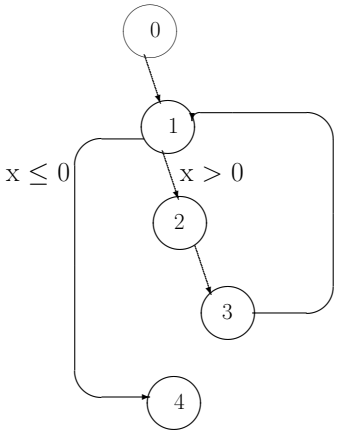

Figura 10.8 Grafo di flusso per l'istruzione `for`

```
// File Do.java
public class Do {
  public static void main(String[] args) {
    /* 0 */ int x = InOut.readInt();
    /* 1 */ do {
    /* 2 */   System.out.println(x);
    /* 3 */   x--;
            }
    /* 4 */ while (x > 0);
  } /* 5 */
}
```

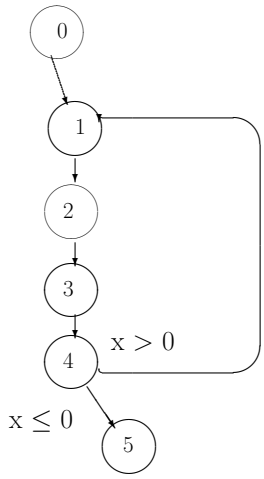

Figura 10.9 Grafo di flusso per l'istruzione `do`

```
// File Return.java
public class Return {
    public static boolean prime(int d) {
        /* 1 */  boolean composite = false;
        /* 2 */  if (d < 0)
        /* 3 */      return false;
        /* 4 */  int i = 1;
        /* 5 */  do {
        /* 6 */      i++;
        /* 7 */      composite = d % i == 0;
                }
        /* 8 */  while(i * i <= d &&  !composite);
        if (composite)
            System.out.println(d + " can be decomposed in: "
                               + i + "*" + d/i);
        else
            System.out.println(d + " is prime");
        /* 9 */  return !composite;
    }

    public static void main(String[] args) {
        int x = InOut.readInt(); // lettura da tastiera
        System.out.println(prime(x));
    }
}
```

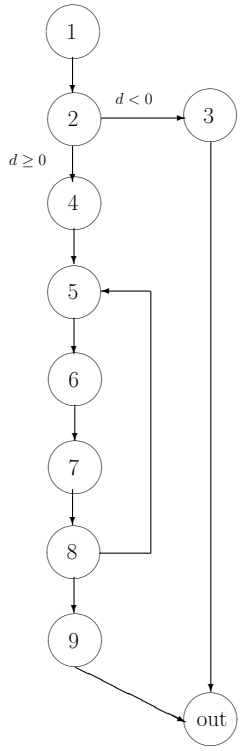

Figura 10.10 Grafo di flusso (parziale) per l'istruzione `return`

```
void f(){
   S1
   // ASSERZIONE
   S2
   // ...
}
```

in cui `S1` e `S2` rappresentano istruzioni qualunque, e `ASSERZIONE` è una formula del prim'ordine. Quest'ultima specifica una proprietà dello stato che la funzione `f()` raggiunge ogni volta che termina l'esecuzione dell'istruzione `S1`. In altri termini, stiamo asserendo che condizione necessaria affinché la funzione `f()` sia corretta è che, dopo l'esecuzione dell'istruzione `S1`, essa raggiunga uno stato in cui vale la condizione espressa da `ASSERZIONE`.

Al fine di esprimere le asserzioni sul programma come commenti, detti appunto *commenti asserzione*, inseriti nel programma stesso, occorre innanzitutto stabilire un metodo per scrivere nel codice i simboli speciali della logica. A questo scopo, useremo nel seguito queste convenzioni:

- Scriveremo `ALL` e `EXISTS` rispettivamente per $\forall$ e $\exists$.
- Scriveremo `&&`, `||` e `!` rispettivamente per $\wedge$, $\vee$ e $\neg$.
- Scriveremo `IMPLIES` e `EQUIV` rispettivamente per $\rightarrow$ e $\equiv$.
- I simboli di predicato e funzione verranno scritti come i corrispondenti simboli in JAVA. Ad esempio, il predicato "$=$" verrà scritto con il simbolo "`==`", il predicato "$\leq$" con il simbolo "`<=`", e così via.

Per usare la logica con lo scopo di scrivere asserzioni come commenti, è evidente che occorre inoltre definire il linguaggio in modo da poter denotare gli oggetti significativi in gioco nello stato del programma. Ciò viene ottenuto definendo il linguaggio della logica in questo contesto nel modo riassunto nella tabella 10.1.

Programma	**Logica**
costante o variabile	simbolo di costante
tipo; funzione (o operatore o simbolo predefinito del linguaggio) che restituisce un valore booleano	simbolo di predicato
funzione (o operatore o simbolo predefinito del linguaggio) che restituisce un valore non booleano	simbolo di funzione
	altri simboli di predicato e funzione (predicati e funzioni "di specifica")

Tabella 10.1 Corrispondenza fra elementi di un programma e simboli della logica

Commentiamo la tabella.

1. Ogni identificatore che denota una costante (predefinita o no) nel programma e ogni identificatore di variabile nel programma è un simbolo di costante nella logica. Si noti che anche i campi di un oggetto sono variabili del programma, e quindi il loro identificatore è una costante della logica.

2. Ogni identificatore di tipo del programma è un simbolo di predicato (in generale ad un argomento) nella logica.

3. Ogni identificatore di funzione del programma o operatore predefinito che restituisce un valore booleano è un simbolo di predicato (di stessa arità) nella logica. Tra gli operatori predefiniti di questo tipo vi sono gli usuali operatori di confronto "`==`"/2, "`<=`"/2 , "`>=`"/2 ecc., che verranno usati in notazione infissa; ad esempio, la formula atomica `==(n,1)` verrà scritta, in modo più naturale, come `n == 1`.

4. Ogni identificatore di funzione del programma o operatore predefinito nel programma che restituisca un valore non booleano è un simbolo di funzione (di stessa arità) nella logica.

I simboli di funzione nella logica corrispondenti agli operatori predefiniti nel programma verranno usati con la notazione infissa; ad esempio, il termine `+(n,1)` verrà scritto, in modo più naturale, come `n+1`.

Si noti che anche l'operatore "`[]`" sugli array e l'operatore "`.`" sugli oggetti verranno scritti in modo coerente con il loro uso nei programmi. Ad esempio, se `v` è un array, il termine che individua la componente di `v` di indice `3` verrà scritto come `v[3]`. Al contrario, applicando formalmente la notazione della logica, "`[]`" sarebbe un simbolo di funzione a due argomenti, e il termine precedente si scriverebbe `[](v,3)`. Analogamente, il campo `info` del record `r` viene denotato dal termine `r.info`, che in logica sarebbe `.(info,r)`.

5. Infine, nella logica, sono definiti altri predicati e altre funzioni, detti "di specifica", che servono appunto nella specifica e che non hanno un corrispettivo nel programma; tra questi assumeremo comunque che esistano i simboli di funzione `result`/0 (costante) e `old`/1, che utilizzeremo più avanti.

Passiamo adesso agli aspetti relativi alla semantica delle asserzioni. Prima di tutto osserviamo che un'asserzione su un programma P è valutata sempre facendo riferimento ad uno stato della computazione di P, quello determinato dal punto in cui il commento asserzione viene inserito. Per determinare il significato di un'asserzione è quindi sufficiente osservare che il dominio, la pre-interpretazione e l'interpretazione di un'asserzione su uno stato di un programma P sono specificati come segue.

- Lo stato della computazione determina in modo ovvio il valore delle costanti e delle variabili del programma, e quindi l'interpretazione dei corrispondenti simboli di costante nella logica; se, ad esempio, il valore della variabile `n` del programma è 5 nello stato S, la pre-interpretazione corrispondente allo stato S assegna 5 alla costante della logica corrispondente a `n`.
- La semantica del linguaggio di programmazione determina il significato dei simboli predefiniti, e quindi l'interpretazione dei corrispondenti simboli di costante, funzione, e predicato nella logica.
- L'interpretazione delle funzioni e dei predicati di specifica è assunta coerente con il significato inteso dato da chi specifica l'asserzione. In particolare, torneremo più avanti sul significato di `result` e `old`.

Esempio 10.3.1 Consideriamo il seguente frammento di programma JAVA, in cui si utilizza la funzione `bubbleSort()` (cfr. paragrafo 10.1):

```
// File ProvaSort.java

public class ProvaSort {
    public static void main(String[] args) {
        int[] v = { 7, 5, 30, 1, 23 };
        BubbleSort.stampaVettore(v);
        int[] w = BubbleSort.bubbleSort(v);;
        BubbleSort.stampaVettore(w);
    }
}
```

Utilizziamo il simbolo di predicato `int`/1, dove il significato inteso di `int(X)` è che "`X` è di tipo intero" e denotiamo con `n` il valore `vett.length`. La formula che asserisce che tutti gli elementi del vettore `v` sono maggiori di `0` è la seguente:

```
ALL X  (int(X)  &&  X >= 0 && X <= n-1)  IMPLIES  v[X] > 0
```

La formula che asserisce che nel vettore è presente una componente con valore pari a quello della variabile `c` è:

```
EXISTS X  int(X)  &&  X >= 0 && X <= n-1 && v[X] == c
```

Infine, la formula che asserisce che il vettore `w` è ordinato in modo crescente è:

```
ALL X  (int(X)  &&  X >= 1 && X <= n-1)  IMPLIES  w[X] > w[X-1]
```

Inserendo le formule come commenti nel codice si ottiene:

```
// File ProvaSort2.java

public class ProvaSort2 {
    public static void main(String[] args) {
        int[] v = { 7, 5, 30, 1, 23 };
        // ALL X  (int(X)  &&  X >= 0 && X <= n-1)  IMPLIES  v[X] > 0
        // EXISTS X  int(X)  &&  X >= 0 && X <= n-1 && v[X] == 23
        BubbleSort.stampaVettore(v);
        int[] w = BubbleSort.bubbleSort(v);;
        // ALL X  (int(X)  &&  X >= 1 && X <= n-1)  IMPLIES  w[X] > w[X-1]
        BubbleSort.stampaVettore(w);
    }
}
```

○

10.3.2 Precondizioni e postcondizioni di una funzione

Nel paragrafo precedente abbiamo discusso dell'utilizzo della logica del prim'ordine per esprimere asserzioni sullo stato di un programma. Un ulteriore interessante utilizzo delle asserzioni riguarda la possibilità di esprimere *precondizioni* e *postcondizioni* associate all'esecuzione di funzioni, o più in generale di sottoprogrammi. Tali condizioni costituiscono la specifica astratta del risultato che la funzione deve calcolare ogni volta che viene invocata in condizioni corrette.

Illustriamo il significato di precondizione e postcondizione di una funzione `f()`.

- La *precondizione* è un'asserzione sullo stato iniziale di ogni esecuzione di `f()`, che stabilisce la condizione che deve essere vera nello stato iniziale di ogni esecuzione di `f()` affinché l'esecuzione stessa sia significativa.

- La *postcondizione* è un'asserzione sullo stato finale di ogni esecuzione della funzione `f()`, che stabilisce la condizione che deve essere vera nello stato finale di ogni esecuzione di `f()` affinché `f()` sia corretta rispetto alle specifiche.

Nel caso in cui la funzione `f()` non abbia particolari precondizioni, useremo come asserzione di precondizione la lettera proposizionale speciale `true`, che viene interpretata a T da ogni interpretazione proposizionale (cfr. definizione 3.3.1).

Poiché la postcondizione riguarda lo stato finale della computazione, abbiamo bisogno in generale di un simbolo che rappresenti il risultato finale, se `f()` restituisce un valore. Questo simbolo è la costante `result`, di cui abbiamo già parlato in precedenza. Inoltre, è possibile che nella postcondizione abbiamo bisogno di riferirci al valore delle variabili nello stato iniziale dell'esecuzione della funzione. Questa è la ragione per cui abbiamo introdotto la funzione `old`/1. In un'asserzione di postcondizione associata ad una certa funzione `f()`, la funzione `old` applicata ad una espressione e, denota il valore di e nello stato iniziale dell'esecuzione di `f()`. Ad esempio, se `alfa` è una variabile utilizzata da una certa funzione `f()`, allora il termine `old(alfa)` nella postcondizione della funzione `f()` denota il valore della variabile `alfa` nello stato iniziale della esecuzione di `f()`.

Gli esempi che seguono mostrano il modo in cui è possibile scrivere le precondizioni e le postcondizioni come commenti alle funzioni JAVA.

Esempio 10.3.2 La seguente funzione calcola la somma di due interi.

```
int Somma(int x, int y)
// commento: calcola la somma di x e y
// pre:  true
// post: result == old(x) + old(y)
{ return x+y; }
```

Si noti che non abbiamo specificato nella precondizione che il tipo delle variabili `x` e `y` deve essere intero. Le condizioni sul tipo delle variabili e dei parametri sono infatti già esplicitate nelle loro dichiarazioni. Ne segue che non ci sono precondizioni per la funzione `Somma()`. ○

Esempio 10.3.3 Consideriamo una funzione che ordina un vettore in maniera crescente effettuando *side effect*, ovvero modificando le sue celle di memoria. La precondizione è che gli elementi del vettore siano tutti diversi fra loro. Le precondizioni e le postcondizioni della funzione possono essere espresse nel seguente modo.

```
void Ordina(int a[], int n)
// commento: ordina il vettore a, che ha n componenti tutte diverse tra loro,
//           in maniera crescente
// pre:  ALL I ALL J (I >= 0 && I <= n-1 && J >=0 && J <= n-1 && I != J)
/                    IMPLIES  a[I] != a[J]
// post: (ALL I (I > 0 && I <= n-1 ) IMPLIES  a[I] > a[I-1])
//       &&
//       (ALL K (K >= 0 && K <= n-1 ) IMPLIES
//        EXISTS L (L >= 0 && L <= n-1 && a[K] == old(a[L])))
//       &&
//       (ALL R (R >= 0 && R <= n-1 ) IMPLIES
//        EXISTS S (S >= 0 && S <= n-1 && old(a[R]) == a[S]))
{ // ......
}
```

Si noti che la postcondizione esprime tre aspetti:

1. che gli elementi del vettore risultante siano ordinati,
2. che tutti gli elementi nel vettore risultante siano presenti nel vettore originario, e
3. che tutti gli elementi del vettore originario siano presenti nel vettore risultante.

○

Esercizio 10.2 [Soluzione a pag. 160] Si consideri la funzione `bubbleSort()` di figura 10.2 che, a differenza della funzione dell'esempio 10.3.3, non effettua *side effect* sul vettore in ingresso. Si esprimano per essa la precondizione e la postcondizione. ○

Esercizio 10.3 [Soluzione a pag. 161] Si consideri la funzione `massimo()` di figura 10.1 e si esprimano per essa la precondizione e la postcondizione. ○

10.3.3 Correttezza parziale e totale di una funzione

In questo paragrafo mostriamo che le nozioni di precondizione e postcondizione ci consentono di definire in modo preciso anche la nozione di correttezza di una funzione rispetto alle specifiche. Si distingue spesso tra due diverse nozioni di correttezza, le cui definizioni sono riportate di seguito.

Definizione 10.3.4 (Correttezza parziale di una funzione) *La funzione* `f()` *è* parzialmente corretta *rispetto alla precondizione σ e alla postcondizione τ se ogni volta che*

1. `f()` *viene eseguita a partire da uno stato iniziale in cui σ è vera, e*
2. *l'esecuzione di* `f()` *termina in uno stato finale s,*

si ha che

3. *τ è vera in s.*

Definizione 10.3.5 (Correttezza totale di una funzione) *La funzione* `f()` *è* totalmente corretta*, o semplicemente* corretta*, rispetto alla precondizione σ e alla postcondizione τ se ogni volta che*

1. `f()` *viene eseguita a partire da uno stato iniziale in cui σ è vera,*

si ha che

2. *l'esecuzione di* `f()` *termina, e*
3. *nello stato finale dell'esecuzione di* `f()` *τ è vera.*

Si noti che la correttezza parziale impone solo che ogni volta che la funzione raggiunge lo stato finale, la postcondizione sia verificata in tale stato. Al contrario, la correttezza totale impone anche che ogni esecuzione della funzione raggiunga effettivamente uno stato finale, e quindi termini. Ne segue che la verifica di correttezza totale implica la dimostrazione di terminazione di ogni esecuzione della funzione che inizi in uno stato in cui vale la precondizione.

Le tecniche formali per la verifica di correttezza parziale e totale dei programmi si basano sulle definizioni appena date. Notiamo che la logica del prim'ordine è insufficiente a rappresentare nozioni come "lo stato finale dell'esecuzione", a cui le definizioni 10.3.4 e 10.3.5 fanno riferimento. Per questo motivo tali nozioni verranno sviluppate nel seguito di questo capitolo facendo riferimento alla logica temporale.

10.4 L'uso della logica temporale per esprimere proprietà di programmi

In questo paragrafo utilizziamo le nozioni di correttezza e terminazione viste nel paragrafo 10.3 nel contesto dei programmi con memoria finita, che abbiamo descritto nel paragrafo 10.2.

In particolare, esprimeremo le proprietà del paragrafo 10.3.3 mediante formule della logica temporale lineare LTL (cfr. capitolo 6). Rispetto alla sintassi stabilita nel paragrafo 10.3, utilizzeremo anche il simbolo di costante PC, il cui significato inteso è "il valore del contatore di programma in un certo stato della computazione". Resta inoltre inteso che, nel momento in cui useremo variabili e quantificatori nelle formule, essi vanno intesi *istanziati* nel dominio di riferimento, in quanto utilizziamo la logica temporale nella sua versione proposizionale.

Terminazione: questa proprietà può essere espressa utilizzando formule temporali del tipo:

$$\sigma \rightarrow F\ \omega \tag{10.1}$$

dove σ è la precondizione della funzione e ω la condizione di terminazione. Quest'ultima può essere espressa nella sequente maniera:

$$PC = ret, \tag{10.2}$$

dove ret è il numero di linea dell'istruzione che contiene l'istruzione di uscita dalla funzione (come `return` o `exit()`), oppure è semplicemente l'ultima istruzione.

Il connettivo temporale F (nel contesto di $F\ \omega$) rappresenta intuitivamente lo "stato finale" nella condizione desiderata; infatti esprime che ci sia un istante nel futuro in cui il valore del contatore di programma ha certificato la terminazione della funzione. Per l'ipotesi del paragrafo 10.1, ovvero che la computazione dell'unità di programma si fermi in corrispondenza delle istruzioni di uscita dalla funzione, di fatto inizia una sequenza infinita di stati in cui il valore del contatore di programma non cambia più. Per questo motivo potremmo esprimere la condizione (10.1) anche con la formula $\sigma \rightarrow F\ G\ \omega$.

Ad esempio, con riferimento alla funzione `massimo()` di figura 10.1, la terminazione viene espressa dalla formula:

$$(\forall X\ X \geq 0\ \wedge\ X \leq vett.length \rightarrow vett[X] > 0) \rightarrow\ F\ PC = 7. \tag{10.3}$$

Notiamo che, nel caso di funzioni con più istruzioni di uscita, al posto della (10.1) dovremmo usare formule del tipo:

$$\sigma \rightarrow \bigvee_{i=1}^{n} (F\ PC = ret_i), \tag{10.4}$$

dove ret_i $(1 \leq i \leq n)$ sono i numeri di linea delle istruzioni che contengono istruzioni di uscita dalla funzione.

Ad esempio, con riferimento alla funzione `prime()` di figura 10.10, che non ha particolari precondizioni, la terminazione viene espressa dalla formula:

$$(F\ PC = 3)\ \vee\ (F\ PC = 9).$$

Correttezza parziale: assumendo un'unica istruzione di uscita dalla funzione, questa proprietà può essere espressa utilizzando formule temporali del tipo:

$$\sigma \rightarrow\ G\ (\omega\ \rightarrow\ \tau), \tag{10.5}$$

dove ω esprime la condizione di terminazione (cfr. (10.2)), σ la precondizione e τ la postcondizione.

Infatti la definizione 10.3.4 richiede che, nell'ipotesi 1 (caratterizzata da σ nell'antecedente dell'implicazione), tutte le volte che si raggiunge (prefisso G) lo stato finale, l'ipotesi 2 (ω) implichi la 3 (τ).

Ad esempio, con riferimento alla funzione `massimo()` di figura 10.1 (cfr. anche esercizio 10.3), la correttezza parziale viene espressa dalla formula:

$$\begin{array}{lll} (\forall X & & X \geq 0\ \wedge\ X \leq vett.length \rightarrow vett[X] > 0) \rightarrow \\ & G & (PC = 7\ \rightarrow \\ & & (\exists X\ X \geq 0\ \wedge\ X \leq vett.length\ \wedge\ vett[X] = result)\ \wedge \\ & & (\forall X\ X \geq 0\ \wedge\ X \leq vett.length \rightarrow result \geq vett[X])). \end{array} \tag{10.6}$$

Correttezza totale: assumendo un'unica istruzione di uscita dalla funzione, questa proprietà può essere espressa utilizzando formule temporali del tipo:

$$\sigma \rightarrow \ (F\ \omega \ \wedge \ G(\omega \ \rightarrow \ \tau)). \tag{10.7}$$

Infatti la correttezza totale (cfr. definizione 10.3.5) consiste nella congiunzione della terminazione (10.1) e della correttezza parziale (10.5), nelle ipotesi date dalle precondizioni.

Ad esempio, con riferimento alla funzione `massimo()` di figura 10.1, la correttezza totale viene espressa dalla formula:

$$\begin{array}{lll} (\forall X & & X \geq 0 \ \wedge \ X \leq vett.length \rightarrow vett[X] > 0) \rightarrow \\ & ((F & PC = 7) \ \wedge \\ & (G & (PC = 7 \ \rightarrow \\ & & (\exists X \ X \geq 0 \ \wedge \ X \leq vett.length \ \wedge \ vett[X] = result) \ \wedge \\ & & (\forall X \ X \geq 0 \ \wedge \ X \leq vett.length \rightarrow result \geq vett[X]))))) \end{array} \tag{10.8}$$

Esercizio 10.4 Fornire le formule corrispondenti a (10.5) e (10.7) per funzioni con più istruzioni di uscita. ◦

Esercizio 10.5 Fornire le formule per la terminazione e la correttezza parziale e totale per la funzione `bubbleSort()` di figura 10.2. ◦

10.5 Aspetti computazionali

In questo paragrafo vogliamo mostrare l'uso di NuSMV per la verifica automatica di proprietà di programmi come quelle presentate nei paragrafi precedenti.

10.5.1 Verifica della terminazione e della correttezza

Per iniziare mostreremo un metodo che trasforma una funzione Java `f()` che soddisfa le condizioni del paragrafo 10.1 in un programma NuSMV `F` atto a verificare in maniera automatica le proprietà menzionate nel paragrafo 10.4. Per la descrizione seguiremo le scelte del paragrafo 10.2.1 sui grafi di flusso, viste nel contesto di LTL.

In questo caso stiamo usando il sistema NuSMV per effettuare il cosiddetto *model checking*, ovvero la verifica che un'implementazione (ovvero la funzione Java, detta *model*) verifichi (*check*) una specifica ((10.1), (10.5), o (10.7)).

Il file NuSMV `F` ha il formato descritto di seguito.

- Vengono dichiarati due moduli:
  1. uno per la descrizione della funzione Java `f()`;
  2. quello obbligatorio (`main`), che funge da cliente del primo e gli fornisce alcune inizializzazioni.
- Nel primo modulo vengono dichiarate le variabili necessarie a descrivere:
  - i parametri formali di `f()`;
  - le variabili locali di `f()` (inclusa quella per il risultato);
  - il contatore di programma di `f()`.

  Ognuna di queste variabili deve appartenere ad un tipo intervallo opportunamente definito. Ad esempio:
  - il contatore di programma assumerà valori compresi fra 1 e il massimo indice di istruzione,
  - le variabili locali assumeranno valori coerenti con le scelte sulla memoria finita usata da `f()`.
- Nel secondo modulo viene dichiarato un oggetto del tipo del primo modulo, di cui si diventa clienti.
- Le assegnazioni iniziali del primo modulo servono per fissare per tutta la computazione, tramite `next()`, i valori dei parametri di `f()` passati per valore, che non devono mai cambiare.

- Le assegnazioni iniziali del secondo modulo servono per fissare il valore iniziale (= 1) del contatore di programma del primo.
- La funzione di transizione del primo modulo esprime la semantica della funzione `f()` (cfr. paragrafo 10.2.1). In particolare, per ogni arco del grafo di flusso di `f()` va previsto un caso:
  - nell'antecedente del caso (a sinistra di "`:`") vanno specificati il valore del PC e l'eventuale condizione di attraversamento (se il nodo ha più archi in uscita);
  - nel conseguente del caso (a destra di "`:`") vanno specificati i valori all'istante successivo di tutte le variabili NuSMV.
- Il secondo modulo non ha funzione di transizione.
- La specifica LTL (definita anche tramite la sezione `DEFINE` del primo modulo) rappresenta la condizione che desideriamo verificare o confutare: (10.1), (10.5) o (10.7).

Una precisazione importante da fare nell'ambito di NuSMV è che esso, in quanto linguaggio proposizionale, non consente facilmente di esprimere condizioni e/o espressioni valide per insiemi di indici. Ad esempio, con riferimento alla riga di codice 4 della funzione `massimo()` di figura 10.1, vorremmo poter usare un'espressione del genere:

```
PC = 4 & vett[i] > res : next(res) = res   & next(PC) = 5 & next(i) = i;
```

che esprime in maniera compatta il fatto che in tale istruzione, quando vale la condizione `vett[i] > res`, si va all'istruzione 5, lasciando inalterati `i` e `res`. Purtroppo la sintassi limitata di NuSMV ci obbliga ad espandere questa espressione prendendo in considerazione tutti i valori di `i` ed ottenendo l'espressione seguente (valida per `i` compreso fra `0` e `2`, cfr. esempio 10.5.1):

```
 PC = 4 & ((i = 0 & vett[0] > res) |
           (i = 1 & vett[1] > res) |
           (i = 2 & vett[2] > res)):
              next(res) = res     & next(PC) = 5 & next(i) = i;
```

La stessa difficoltà si riscontra nel caso di formule quantificate, quali le precondizioni e le postcondizioni. Ad esempio,siamo costretti ad espandere la precondizione della funzione `massimo()` di figura 10.1 (cfr. formula (10.18)) nel seguente modo:

```
m.vett[0] > 0 & m.vett[1] > 0 & m.vett[2] > 0
```

Esempio 10.5.1 Con riferimento alla funzione funzione `massimo()` della figura 10.1, abbiamo descritto in precedenza le seguenti proprietà:

terminazione: (10.3),

correttezza parziale: (10.6),

correttezza totale: (10.8).

Riportiamo il file NuSMV ottenuto secondo le regole viste in precedenza, che contiene la traduzione della funzione `massimo()` e la richiesta di verifica delle varie condizioni nelle seguenti ipotesi:

- vettore `vett` di input di tre elementi,
- elementi di `vett` compresi fra -10 e 10,
- variabili locali `i` e `res` opportunamente limitate.

```
-- Time-stamp: "2006-08-02 12:25:34 cadoli"
-- File: massimo.smv
-- Descrizione: correttezza funzione Java che trova il massimo

MODULE massimo
-- questo modulo descrive la funzione Java
-- public static int massimo(int[] vett)
-- ASSUNZIONI:
--   la lunghezza del vettore è 3
--   gli elementi del vettore appartengono all'intervallo -10..10
VAR
```

```
-- variabili per i parametri formali Java
  vett : array 0..2 of -10..10;
-- variabili per le variabili locali Java
  i    : 0..3;
  res  : -10..10;
-- program counter
  PC : 1..7;
ASSIGN
-- vett è passato per valore (i suoi elementi non cambiano)
  next(vett[0]) := vett[0];
  next(vett[1]) := vett[1];
  next(vett[2]) := vett[2];
DEFINE
  PRE := vett[0] > 0 & vett[1] > 0 & vett[2] > 0;
  POST:= (res = vett[0] | res = vett[1] | res = vett[2]) &
         (res >= vett[0] & res >= vett[1] & res >= vett[2]);
  TERM:= PC = 7;
TRANS
  case
    PC = 1     : next(res) = 0       & next(PC) = 2 & next(i) = i;
    PC = 2     : next(res) = res     & next(PC) = 3 & next(i) = 0;
    PC = 3 &
    i < 3      : next(res) = res     & next(PC) = 4 & next(i) = i;
    PC = 3 &
    i >= 3     : next(res) = res     & next(PC) = 7 & next(i) = i;
    PC = 4 & ((i = 0 & vett[0] > res) |
              (i = 1 & vett[1] > res) |
              (i = 2 & vett[2] > res)):
                next(res) = res     & next(PC) = 5 & next(i) = i;
    PC = 4 & ((i = 0 & vett[0] <= res) |
              (i = 1 & vett[1] <= res) |
              (i = 2 & vett[2] <= res)):
                next(res) = res     & next(PC) = 6 & next(i) = i;
    PC = 5 & i = 0 :
                next(res) = vett[0] & next(PC) = 6 & next(i) = i;
    PC = 5 & i = 1 :
                next(res) = vett[1] & next(PC) = 6 & next(i) = i;
    PC = 5 & i = 2 :
                next(res) = vett[2] & next(PC) = 6 & next(i) = i;
    PC = 6     : next(res) = res     & next(PC) = 3 & next(i) = i + 1;
    PC = 7     : next(res) = res     & next(PC) = 7 & next(i) = i;
  esac
-- end MODULE massimo

MODULE main
VAR
  m: massimo;
ASSIGN
-- inizializzazione program counter
  init(m.PC) := 1;
-- end MODULE main

LTLSPEC
-- TERMINAZIONE: PRE -> F TERM
-- m.PRE -> F m.TERM
-- CORRETTEZZA PARZIALE: PRE -> G(TERM -> POST)
-- m.PRE -> G(m.TERM -> m.POST)
-- CORRETTEZZA TOTALE: PRE -> (F TERM & G(TERM -> POST))
m.PRE -> (F m.TERM & G(m.TERM -> m.POST))
```

Le tre condizioni menzionate vengono verificate individualmente in alcuni secondi ciascuna. ○

È interessante notare che il sistema trova applicazioni anche in altri casi. Di seguito mostriamo alcuni esempi relativi a situazioni frequenti:

- programmi JAVA con errori,

- specifiche LTL scorrette.

Esempio 10.5.2 [continuazione dell'esempio 10.5.1] Con riferimento alla funzione `massimo()` di figura 10.1, immaginiamo di omettere per errore dal codice JAVA la riga 6, impedendo così alla funzione di terminare. Il file NUSMV deve essere opportunamente modificato, eliminando ogni riferimento all'istruzione 6 (ad esempio, dopo l'istruzione 5 deve seguire la 3).

Il sistema NUSMV costruisce una traccia dell'esecuzione del programma costituita da opportuni valori di input che dimostrano la non terminazione, ovvero il mancato raggiungimento dell'istruzione 7:

```
-- specification  (((m.vett[0] > 0 & m.vett[1] > 0) & m.vett[2] > 0) ->  F m.PC = 7)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  m.vett[0] = 5
  m.vett[1] = 5
  m.vett[2] = 5
  m.i = 0
  m.res = -10
  m.PC = 1
-> State: 1.2 <-
  m.res = 0
  m.PC = 2
-> State: 1.3 <-
  m.PC = 3
-> State: 1.4 <-
  m.PC = 4
-> State: 1.5 <-
  m.PC = 5
-- Loop starts here
-> State: 1.6 <-
  m.res = 5
  m.PC = 3
-> State: 1.7 <-
  m.PC = 4
-> State: 1.8 <-
  m.PC = 3
```

∘

Esempio 10.5.3 [continuazione dell'esempio 10.5.1] Supponiamo ora di scrivere in maniera errata la condizione di correttezza parziale (10.6), omettendo la precondizione dall'antecedente ("σ" in "$\sigma \rightarrow G(\omega \rightarrow \tau)$") dell'implicazione. In tal caso la funzione `massimo()` è errata, a causa dell'istruzione 1. Il sistema NUSMV determina un controesempio che consiste in un vettore di input costituito da elementi tutti negativi, per cui il valore trovato non soddisfa la postcondizione (10.19):

```
-- specification   G (m.PC = 7 -> (((m.res = m.vett[0] | m.res = m.vett[1]) |
m.res = m.vett[2]) &
((m.res >= m.vett[0] & m.res >= m.vett[1]) & m.res >= m.vett[2])))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  m.vett[0] = -3
  m.vett[1] = -3
  m.vett[2] = -3
  m.i = 0
  m.res = -10
  m.PC = 1
-> State: 1.2 <-
  m.res = 0
  m.PC = 2
-> State: 1.3 <-
  m.PC = 3
-> State: 1.4 <-
  m.PC = 4
```

```
-> State: 1.5 <-
  m.PC = 6
-> State: 1.6 <-
  m.i = 1
  m.PC = 3
-> State: 1.7 <-
  m.PC = 4
-> State: 1.8 <-
  m.PC = 6
-> State: 1.9 <-
  m.i = 2
  m.PC = 3
-> State: 1.10 <-
  m.PC = 4
-> State: 1.11 <-
  m.PC = 6
-> State: 1.12 <-
  m.i = 3
  m.PC = 3
-- Loop starts here
-> State: 1.13 <-
  m.PC = 7
-- Loop starts here
-> State: 1.14 <-
-> State: 1.15 <-
```

○

10.5.2 Uso del sistema come solutore

In questo paragrafo vogliamo mostrare un ulteriore uso di NuSMV, in particolare come risolutore di problemi. A differenza del paragrafo precedente 10.5.1, non vogliamo confrontare una specifica LTL con un'implementazione Java, ma vogliamo direttamente trovare la soluzione di un problema ϕ data la sua precondizione σ e la sua postcondizione τ, per una particolare istanza i di ϕ.

Ciò può essere ottenuto chiedendo di trovare un'interpretazione dei simboli o di ϕ che ne rappresentano l'output, tale che valga la seguente proprietà:

$$i \ \wedge \ o \models (\sigma \ \rightarrow \ \tau). \tag{10.9}$$

In questa maniera abbracciamo un paradigma di programmazione che si va notevolmente diffondendo, noto col nome di *programmazione a vincoli* (cfr. capitolo **??**). In particolare stiamo usando il sistema NuSMV per effettuare il cosiddetto *model finding*, ovvero la richiesta di trovare (*find*) un'assegnazione alle variabili di output (ovvero o) che verifichi la specifica (10.9).

In questo caso il file NuSMV è preparato secondo il seguente formato.

- Vengono dichiarati due moduli:
  1. uno per la descrizione del problema ϕ;
  2. quello obbligatorio (`main`), che funge da cliente del primo e gli fornisce alcune inizializzazioni.
- Nel primo modulo vengono dichiarate le variabili necessarie a descrivere:
  - l'input di ϕ;
  - l'output di ϕ.

  Ognuna di queste variabili deve appartenere ad un tipo intervallo opportunamente definito.
- Nel secondo modulo viene dichiarato un oggetto del tipo del primo modulo, di cui si diventa clienti.
- Le assegnazioni iniziali del primo modulo servono per fissare, tramite `next()`, per tutta la computazione i valori dell'input di ϕ.
- Le assegnazioni iniziali del secondo modulo servono per fissare i valori dell'input di ϕ all'istante iniziale e per affermare che il risultato, una volta calcolato, non cambia più.

- La funzione di transizione è assente per entrambi i moduli, in quanto non stiamo verificando proprietà dinamiche.
- La specifica LTL rappresenta la condizione che desideriamo verificare, ovvero una formula del tipo (10.9). In particolare, desideriamo trovare un *controesempio* al fatto che le precondizioni implichino le postcondizioni.

La tabella 10.2 rappresenta in maniera compatta le differenze dei vari elementi di NuSMV nei due possibili utilizzi del sistema.

		Model checking (par. 10.5.1)	**Solutore** (par. 10.5.2)
modulo generico	`VAR`	input & output; variabili locali JAVA; PC	input & output
	`ASSIGN`	input (per valore) non cambia	input non cambia
	`TRANS`	traduzione funzione JAVA	–
modulo `main`	`VAR`	cliente modulo generico	cliente modulo generico
	`ASSIGN`	PC	istanza (input)
	`TRANS`	–	–
	`LTLSPEC`	verifica proprietà: (10.1), (10.5) o (10.7)	soluzione: (10.9)

Tabella 10.2 Uso di NuSMV per *model checking* e come solutore

Esempio 10.5.4 [continuazione dell'esempio 10.5.1] In questo caso abbiamo la precondizione (10.18) e la postcondizione (10.19-10.20).

Il file NuSMV per il vettore di input `[7,3,1]` è il seguente.

```
-- Time-stamp: "2006-06-14 12:24:01 cadoli"
-- File: solvemax.smv
-- Descrizione: NuSMV usato come solver per il problema di trovare il massimo in un array

MODULE massimo
VAR
-- variabili per l'input
  vett : array 0..2 of 0..20;
-- variabili per l'output
  result  : 0..20;
ASSIGN
-- i dati di input non cambiano più
  next(vett[0]) := vett[0];
  next(vett[1]) := vett[1];
  next(vett[2]) := vett[2];
-- end MODULE massimo

MODULE main
VAR
  m: massimo;
ASSIGN
-- inizializzazione dati di input (codifica di un'istanza)
  init(m.vett[0]) := 7;
  init(m.vett[1]) := 3;
  init(m.vett[2]) := 1;
-- result viene calcolato e non cambia più
  next(m.result) := m.result;

LTLSPEC
-- formula da verificare: ! (precondizione -> postcondizione).
-- Se il problema è risolubile, la postcondizione è vera, trova un
-- controesempio, che è la soluzione
```

```
    !(
-- elementi del vettore positivi
       (m.vett[0] > 0 & m.vett[1] > 0 & m.vett[2] > 0) ->
-- result è presente nel vettore
       (m.result = m.vett[0] | m.result = m.vett[1] | m.result = m.vett[2]) &
-- result è maggiore o uguale di ogni elemento del vettore
       (m.result >= m.vett[0] & m.result >= m.vett[1] & m.result >= m.vett[2])
     );
```

La soluzione (valore 7, massimo nel vettore) viene rapidamente determinata, come mostrato dall'output del sistema:

```
-- specification  !(((m.vett[0] > 0 & m.vett[1] > 0) & m.vett[2] > 0) -> (((m.result = m.vett[0] | m.result = m.v
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  m.vett[0] = 7
  m.vett[1] = 3
  m.vett[2] = 1
  m.result = 7
-> State: 1.2 <-
```

○

Esempio 10.5.5 In questo caso riprendiamo l'esempio dell'ordinamento del vettore (cfr. figura 10.2 ed esercizi 10.1, 10.2, 10.5). Per semplicità dichiariamo un modulo unico e, a differenza dell'esempio precedente, dichiariamo le variabili *interne* `perm_0, perm_1, perm_2`, che servono per rappresentare la permutazione che caratterizza l'ordinamento del vettore.

Per quanto riguarda le nuove variabili `perm_0, perm_1, perm_2`, dobbiamo specificare che:

- una volta assegnate, non cambiano più (tramite `next()` in `ASSIGN`), e che
- l'output (`result[]`) viene definito (in `DEFINE`) mediante esse e il vettore di input (`vett[]`).

Il file NuSMV per il vettore di input `[10,2,3]` è il seguente.

```
-- Time-stamp: "2006-06-14 13:33:35 cadoli"
-- File: solvesort.smv
-- Descrizione: NuSMV usato come solver per il problema di ordinare un vettore
-- Si puo' usare con bounded model checking (BMC e bmc_length = 0)
MODULE main
VAR
-- variabili per l'input
  vett : array 0..2 of 0..20;
-- variabili per lo spazio di ricerca
  perm_0 : 0..2;
  perm_1 : 0..2;
  perm_2 : 0..2;
-- non posso definirlo come array, perche' non posso usare array come elementi
-- di array :-(
--  perm : array 0..2 of 0..2;
ASSIGN
-- inizializzazione dati di input
  init(vett[0]) := 10;
  init(vett[1]) := 2;
  init(vett[2]) := 3;
-- i dati di input non cambiano piu'
  next(vett[0]) := vett[0];
  next(vett[1]) := vett[1];
  next(vett[2]) := vett[2];
-- perm viene calcolato e non cambia piu'
  next(perm_0) := perm_0;
  next(perm_1) := perm_1;
  next(perm_2) := perm_2;
```

```
DEFINE
-- definizione di result (va qui e non in VAR per efficienza)
-- purtroppo non si puo' scrivere
--     result[0] = vett[perm_0];
--     result[1] = vett[perm_1];
--     result[2] = vett[perm_2];
  result[0] := case
                 perm_0 = 0 : vett[0];
                 perm_0 = 1 : vett[1];
                 perm_0 = 2 : vett[2];
               esac;
  result[1] := case
                 perm_1 = 0 : vett[0];
                 perm_1 = 1 : vett[1];
                 perm_1 = 2 : vett[2];
               esac;
  result[2] := case
                 perm_2 = 0 : vett[0];
                 perm_2 = 1 : vett[1];
                 perm_2 = 2 : vett[2];
               esac;
-- end MODULE main

LTLSPEC
-- formula da verificare: (! postcondizione).
-- Se il problema e' risolubile, la postcondizione e' vera, trova un
-- controesempio, che e' la soluzione
 !(
-- 1. elements of the output array are sorted
     (result[2] >= result[1] & result[1] >= result[0]) &

-- 2. all elements of the indexes occur in perm --> perm is a permutation
     ((perm_0 = 0) + (perm_1 = 0) + (perm_2 = 0) >= 1) &
     ((perm_0 = 1) + (perm_1 = 1) + (perm_2 = 1) >= 1) &
     ((perm_0 = 2) + (perm_1 = 2) + (perm_2 = 2) >= 1) &

-- 3. perm is alldifferent --> perm is a permutation
     (perm_0 != perm_1 & perm_0 != perm_2 & perm_1 != perm_2)

-- equivalent specifications: 1+2, 1+3, 1+2+3
   );
```

La soluzione (vettore `[2,3,10]`, ordinato) viene rapidamente determinata, come mostrato dall'output del sistema:

```
-- specification  !(((((result[2] >= result[1] & result[1] >= result[0]) & (perm_0 = 0) + (perm_1 = 0) + (perm_2
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  vett[0] = 10
  vett[1] = 2
  vett[2] = 3
  perm_0 = 1
  perm_1 = 2
  perm_2 = 0
  result[2] = 10
  result[1] = 3
  result[0] = 2
-> State: 1.2 <-
```

∘

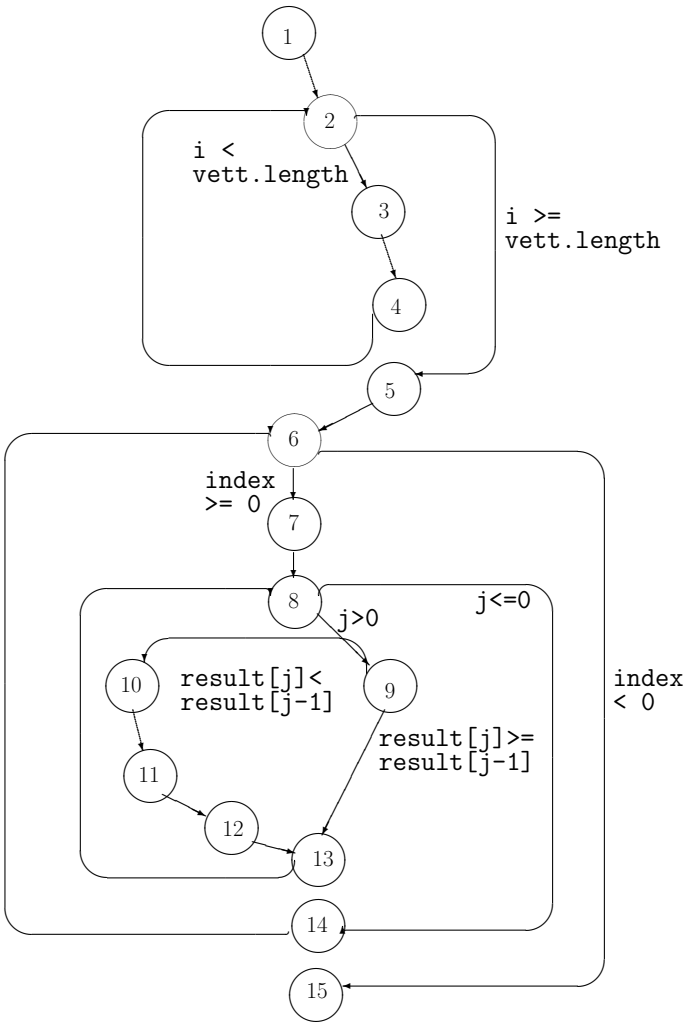

Figura 10.11 Grafo di flusso per la funzione `bubbleSort()` di figura 10.2

10.6 Nota bibliografica

Per quanto riguarda l'uso della logica per esprimere proprietà di programmi, ed in particolare per la proprietà di correttezza, si rimanda a [21, 26, 27]. In questi testi le questioni riguardanti la terminazione e la correttezza vengono affrontate nel contesto di programmi senza limitazioni di memoria.

10.7 Esercizi proposti

Esercizio 10.6 [Soluzione a pag. **??**] Riprendendo l'esempio dell'ordinamento del vettore (cfr. figura 10.2, esercizi 10.1, 10.2, 10.5, ed esempio 10.5.5), fornire il file NuSMV per la verifica della terminazione e della correttezza parziale e totale della funzione Java `bubbleSort()`.

10.8 Soluzione di alcuni esercizi

Soluzione esercizio 10.1. Il grafo di flusso per l'unità di programma `bubbleSort()` di figura 10.2 è riportato in figura 10.11.

○

Soluzione esercizio 10.2. Supponiamo che la precondizione sia:

$$\forall XY \quad (X \geq 0 \ \wedge X \leq vett.length - 1 \ \wedge \ Y \geq 0 \ \wedge \ Y \leq vett.length - 1 \ \wedge \ X \neq Y) \\ \rightarrow vett[X] \neq vett[Y], \tag{10.10}$$

ovvero che gli elementi di `vett[]` siano tutti diversi fra loro.

A differenza dell'esempio 10.3.3, ora facciamo riferimento al vettore `result[]`, che in logica del prim'ordine rappresentiamo come simbolo di funzione $result/1$.

Come nell'esempio, esprimiamo i tre aspetti della postcondizione:

- che gli elementi del vettore `result[]` siano ordinati (10.11),
- che tutti gli elementi nel vettore `result[]` siano presenti nel vettore `vett[]` (10.12), e
- che tutti gli elementi del vettore `vett[]` siano presenti nel vettore `result[]` (10.13).

$$\forall X \quad (X > 0 \ \wedge \ X \leq vett.length - 1) \ \rightarrow result[X] > result[X-1] \quad \wedge \tag{10.11}$$

$$\forall X \quad (X \geq 0 \ \wedge \ X \leq vett.length - 1) \ \rightarrow \\ \exists Y \ (Y \geq 0 \ \wedge \ Y \leq vett.length - 1 \ \wedge \ result[X] = old(vett[Y])) \quad \wedge \tag{10.12}$$

$$\forall X \quad (X \geq 0 \ \wedge \ X \leq vett.length - 1) \ \rightarrow \\ \exists Y \ (Y \geq 0 \ \wedge \ Y \leq vett.length - 1 \ \wedge \ result[Y] = old(vett[X])). \tag{10.13}$$

Notiamo la necessità delle condizioni (10.12) e (10.13): se ad esempio eliminiamo la prima, potremmo avere come input `vett = { 7, 5, 3, 1, -3 }` e come output `result = { -3, 1, 0, 5, 7 }`, ovvero il valore `0` occorre in `result[]` ma non in `vett[]`.

È chiaro che la precondizione (10.10) non è affatto necessaria ai fini della correttezza della funzione `bubbleSort()`, e in effetti è stata posta solamente per avere postcondizioni semplici. Per esprimere la postcondizione in assenza di precondizione (ovvero quando quest'ultima vale `true`) abbiamo bisogno di un vettore ausiliario `perm[]` che rappresenti una permutazione degli indici del vettore `vett[]` di ingresso. Ad esempio, se `vett = { 7, -5, -3, 1, -3 }` una permutazione di indici che lo ordina in maniera non decrescente è `perm = { 1, 2, 4, 3, 0 }`. La permutazione non è unica: ad esempio è valida anche `perm = { 1, 4, 2, 3, 0 }`. La nuova postcondizione è riportata di seguito.

$$\forall X \quad (X > 0 \ \wedge \ X \leq vett.length - 1) \ \rightarrow result[X] \geq result[X-1] \quad \wedge \tag{10.14}$$

$$\forall X \quad (X \geq 0 \ \wedge \ X \leq vett.length - 1) \ \rightarrow result[X] = vett[perm[X]] \quad \wedge \tag{10.15}$$

$$\forall X \quad (X \geq 0 \ \wedge \ X \leq vett.length - 1) \ \rightarrow \\ \exists Y \ (Y \geq 0 \ \wedge \ Y \leq vett.length - 1 \ \wedge \ perm[X] = Y) \quad \wedge \tag{10.16}$$

$$\forall X \quad (X \geq 0 \ \wedge \ X \leq vett.length - 1) \ \rightarrow \\ \exists Y \ (Y \geq 0 \ \wedge \ Y \leq vett.length - 1 \ \wedge \ perm[Y] = X) \tag{10.17}$$

Commentiamo i vari aspetti della postcondizione:

- gli elementi del vettore `result[]` devono essere ordinati in maniera non decrescente (10.14),
- gli elementi nel vettore `result[]` vengono ottenuti tramite quelli del vettore `vett[]`, permutati attraverso `perm[]` (10.15),
- il vettore `perm[]` contiene solo indici del vettore `vett[]` (10.16),
- gli indici del vettore `vett[]` sono presenti nel vettore `perm[]` (10.17).

Per concludere, notiamo che nelle formule (10.14-10.17) il vettore `perm[]` occorre *libero*, ed andrebbe quantificato esistenzialmente, entrando in tal modo nella sfera della *logica del second'ordine* (cfr. capitolo 5). ◦

Soluzione esercizio 10.3. Sia *vett* il vettore di ingresso.

precondizione: gli elementi di *vett* sono positivi:

$$\forall X \qquad X \geq 0 \ \wedge \ X < vett.length \rightarrow vett[X] > 0. \tag{10.18}$$

postcondizione: *result* occorre in *vett* (10.19) e *result* è magggiore o uguale di tutti gli elementi di *vett* (10.20):

$$(\exists X \qquad X \geq 0 \ \wedge \ X < vett.length \ \wedge \ vett[X] = result) \ \wedge \tag{10.19}$$
$$(\forall X \qquad X \geq 0 \ \wedge \ X < vett.length \rightarrow result \geq vett[X]). \tag{10.20}$$

∘

Soluzione esercizio 10.5. Per la funzione `bubbleSort()` di figura 10.2 possiamo considerare due interpretazioni:

1. con la precondizione (10.10) e le postcondizioni (10.11, 10.12, 10.13);
2. senza precondizione e con le postcondizioni (10.14, 10.15, 10.16, 10.17).

Il file NuSMV per la versione con precondizione (per semplicità, con un solo modulo) è riportata di seguito.

```
-- Time-stamp: "2006-08-03 00:12:08 cadoli"
-- File: bubbleSortPRECOND.smv
-- Descrizione: correttezza funzione Java che ordina un vettore
-- VERSIONE CON PRECONDIZIONI
--       n = vett.length
-- pre:  ALL I ALL J (I >= 0 && I <= n-1 && J >= 0 && J <= n-1 && I != J)
--                   IMPLIES  vett[I] != vett[J]
--       gli elementi del vettore vett sono tutti diversi fra loro
-- post: (ALL I (I > 0 && I <= n-1) IMPLIES  result[I] > result[I-1])
-- POST1: gli elementi del vettore result sono ordinati
--       &&
--       (ALL K (K >= 0 && K <= n-1) IMPLIES
--        EXISTS L (L >= 0 && L <= n-1 && result[K] == old(vett[L])))
-- POST2: gli elementi nel vettore result sono presenti nel vettore vett
--       &&
--       (ALL R (R >= 0 && R <= n-1) IMPLIES
--        EXISTS S (S >= 0 && S <= n-1 && old(vett[R]) == result[S]))
-- POST3: gli elementi nel vettore vett sono presenti nel vettore result

MODULE main
-- questo modulo descrive la funzione Java
--    public static int[] bubbleSort(int[] vett)
-- e il suo cliente
-- ASSUNZIONI:
--   la lunghezza del vettore (vett.length) è 3
--   gli elementi del vettore appartengono all'intervallo 0..2 (0..MAXINT)
VAR
-- variabili per i parametri formali Java
  vett : array 0..2 of 0..2; -- 0..MAXINT
-- variabili per le variabili locali Java
  i    : 0..3;                --  0..vett.length
  j    : 0..3;                --  0..vett.length
  index: -1..3;               -- -1..vett.length, cfr. PC = 14
  temp : 0..2;                --  0..MAXINT
  result: array 0..2 of 0..2; --  0..MAXINT
-- program counter
  PC : 1..15;                 -- 1..MAXPC
ASSIGN
-- vett è passato per valore (gli elementi non cambiano)
```

```
  next(vett[0]) := vett[0];
  next(vett[1]) := vett[1];
  next(vett[2]) := vett[2];
-- inizializzazione program counter
  init(PC) := 1;
DEFINE
-- definizione di PRE, POST, TERM
  PRE  := vett[0] != vett[1] & vett[0] != vett[2] & vett[1] != vett[2];
  POST1:= result[2] > result[1] & result[1] > result[0];
  POST2:= (result[0] = vett[0] | result[0] = vett[1] | result[0] = vett[2]) &
          (result[1] = vett[0] | result[1] = vett[1] | result[1] = vett[2]) &
          (result[2] = vett[0] | result[2] = vett[1] | result[2] = vett[2]);
  POST3:= (vett[0] = result[0] | vett[0] = result[1] | vett[0] = result[2]) &
          (vett[1] = result[0] | vett[1] = result[1] | vett[1] = result[2]) &
          (vett[2] = result[0] | vett[2] = result[1] | vett[2] = result[2]);
  POST := POST1 & POST2 & POST3;
  TERM := PC = 15;
TRANS
  case
    PC = 1 :
      next(PC) = 2         & next(i) = 0       & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 2 & i < 3  :
      next(PC) = 3         & next(i) = i       & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 2 & i >= 3  :
      next(PC) = 5         & next(i) = i       & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 3 & i = 0  :
      next(PC) = 4         & next(i) = i       & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = vett[0]   & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 3 & i = 1  :
      next(PC) = 4         & next(i) = i       & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = vett[1] &
      next(result[2]) = result[2];
    PC = 3 & i = 2  :
      next(PC) = 4         & next(i) = i       & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = vett[2];
    PC = 4 :
      next(PC) = 2         & next(i) = i + 1   & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 5 :
      next(PC) = 6         & next(i) = i       & next(j) = j &
      next(index) = 3-1    & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 6 & index >= 0  :
      next(PC) = 7         & next(i) = i       & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 6 & index < 0  :
      next(PC) = 15        & next(i) = i       & next(j) = j &
```

```
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 7 :
  next(PC) = 8         & next(i) = i       & next(j) = 3-1 &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 8 & j > 0 :
  next(PC) = 9         & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 8 & j <= 0 :
  next(PC) = 14        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 9 &
((j = 1 & result[1] < result[0]) | (j = 2 & result[2] < result[1])) :
  next(PC) = 10        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 9 &
((j = 1 & result[1] >= result[0]) | (j = 2 & result[2] >= result[1])) :
  next(PC) = 13        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 10 & j = 1 :
  next(PC) = 11        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = result[1] &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 10 & j = 2 :
  next(PC) = 11        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = result[2] &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 11 & j = 1 :
  next(PC) = 12        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[0] &
  next(result[2]) = result[2];
PC = 11 & j = 2 :
  next(PC) = 12        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[1];
PC = 12 & j = 1 :
  next(PC) = 13        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = temp & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 12 & j = 2 :
  next(PC) = 13        & next(i) = i       & next(j) = j &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = temp &
  next(result[2]) = result[2];
PC = 13 :
  next(PC) = 8         & next(i) = i   & next(j) = j - 1 &
  next(index) = index  & next(temp) = temp &
  next(result[0]) = result[0] & next(result[1]) = result[1] &
  next(result[2]) = result[2];
PC = 14 :
```

```
      next(PC) = 6         & next(i) = i   & next(j) = j &
      next(index) = index - 1 & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
    PC = 15 :
      next(PC) = 15        & next(i) = i   & next(j) = j &
      next(index) = index  & next(temp) = temp &
      next(result[0]) = result[0] & next(result[1]) = result[1] &
      next(result[2]) = result[2];
  esac
-- end MODULE main

LTLSPEC
-- TERMINAZIONE
-- PRE -> F TERM
-- CORRETTEZZA PARZIALE
-- PRE -> G (TERM -> POST)
-- CORRETTEZZA TOTALE
 PRE -> (F TERM & G (TERM -> POST))
```

Il file NuSMV per la versione senza precondizione presenta alcune differenze, evidenziate dalla tabella 10.3.

	`bubbleSortPRECOND.smv`	`bubbleSort.smv`
`VAR`	JAVA: parametri formali (`vett[]`) e variabili locali (`i`, `j`, `index`, `temp`, `result`); `PC`.	JAVA: parametri formali (`vett[]`) e variabili locali (`i`, `j`, `index`, `temp`); `PC`; `perm[]` (con (10.16)).
`ASSIGN`	input (`vett[]`, per valore) non cambia; `PC = 1`.	idem + `perm[]` non cambia.
`DEFINE`	`PRE`, `POST`, `TERM`.	idem + `result[]` (con (10.15)).
`TRANS`	traduzione funzione JAVA.	idem.
`LTLSPEC`	verifica proprietà: (??), (??) o (??).	idem.

Tabella 10.3 File NuSMV per la verifica di `bubbleSort()` con e senza precondizioni

La parte del file differente dalla precedente (fino a `DEFINE` incluso e `TRANS` escluso) è riportata di seguito.

```
-- Time-stamp: "2006-07-27 21:28:34 cadoli"
-- File: bubbleSort.smv
-- Descrizione: correttezza funzione Java che ordina un vettore
-- VERSIONE SENZA PRECONDIZIONI
--       n = vett.length
-- pre:  TRUE
-- post: (ALL I (I > 0 && I <= n-1 ) IMPLIES  result[I] >= result[I-1])
-- POST1:gli elementi del vettore result sono ordin. in maniera non decr.
--       &&
--       (ALL X (X >= 0 && X <= n-1) IMPLIES result[X] = vett[perm[X]]
-- POST2:gli elementi nel vettore result vengono ottenuti tramite quelli
--       del vettore vett, permutati attraverso perm
--       &&
--       (ALL X (X >= 0 && X <= n-1) IMPLIES
--        EXISTS Y (Y >= 0 && Y <= n-1 && perm[X] = Y)
-- POST3:il vettore perm contiene solo indici del vettore vett
--       &&
--       (ALL X (X >= 0 && X <= n-1) IMPLIES
--        EXISTS Y (Y >= 0 && Y <= n-1 && perm[Y] = X)
```

```
-- POST4:gli indici del vettore vett sono presenti nel vettore perm

MODULE main
-- questo modulo descrive la funzione Java
--    public static int[] bubbleSort(int[] vett)
-- ASSUNZIONI:
--   la lunghezza del vettore (vett.length) è 3
--   gli elementi del vettore appartengono all'intervallo 0..10 (0..MAXINT)
VAR
-- variabili per i parametri formali Java
  vett : array 0..2 of 0..10; -- 0..MAXINT
-- variabili per le variabili locali Java
  i    : 0..3;                -- 0..vett.length
  j    : 0..3;                -- 0..vett.length
  index: -1..3;               -- -1..vett.length, cfr. PC = 14
  temp : 0..10;               -- 0..MAXINT
-- program counter
  PC : 1..15;                 -- 1..MAXPC
-- variabili per lo spazio di ricerca (interne, servono per la specifica)
-- IMPLEMENTA POST3
  perm : array 0..2 of 0..2;  -- 0..vett.length
ASSIGN
-- vett è passato per valore (gli elementi non cambiano)
  next(vett[0]) := vett[0];
  next(vett[1]) := vett[1];
  next(vett[2]) := vett[2];
-- inizializzazione program counter
  init(PC) := 1;
-- perm viene calcolato e non cambia più (non è una variabile Java)
  next(perm[0]) := perm[0];
  next(perm[1]) := perm[1];
  next(perm[2]) := perm[2];
DEFINE
-- definizione di result (va qui per efficienza)
-- IMPLEMENTA POST2
  result[0] := case
                 perm[0] = 0 : vett[0];
                 perm[0] = 1 : vett[1];
                 perm[0] = 2 : vett[2];
               esac;
  result[1] := case
                 perm[1] = 0 : vett[0];
                 perm[1] = 1 : vett[1];
                 perm[1] = 2 : vett[2];
               esac;
  result[2] := case
                 perm[2] = 0 : vett[0];
                 perm[2] = 1 : vett[1];
                 perm[2] = 2 : vett[2];
               esac;
-- definizione di PRE, POST, TERM
  PRE  := 1;
  POST1:= result[2] >= result[1] & result[1] >= result[0];
  POST4:= (perm[0] = 0 | perm[1] = 0 | perm[2] = 0) &
          (perm[0] = 1 | perm[1] = 1 | perm[2] = 1) &
          (perm[0] = 2 | perm[1] = 2 | perm[2] = 2);
  POST := POST1 & POST4;
  TERM := PC = 15;
```

La figura 10.12 mostra che la verifica delle proprietà di correttezza richiede, anche per piccoli domini, alcuni minuti di tempo macchina. La versione senza precondizioni sembra scalare meglio all'aumentare della dimensione del dominio (probabilmente a causa del minore numero di stati previsti).

○

```
-- Time-stamp: "2006-08-02 12:27:38 cadoli"
-- dati ottenuti con VAIO VGN-X505 (1.1 GHz, 508Mb)
-- vett.length = 3

============+==================+===============+===============+
            |  massimo()       | bubbleSort()  |  bubbleSort() |
            |                  |   PRECOND     |               |
============+==================+===============+===============+
elementi di | -10..10  -20..20 | 0..4   0..5   | 0..4   0..5   |
vett[] in   |    (a)      (b)  |  (c)    (d)   |  (e)    (f)   |
============+==================+===============+===============+
TERMINAZIONE|   13"      3'55" |    6"   1'10" | 1'11"   2'10" |
CORRETT.PAR.|   21"      4'28" | 3'10"  17'01" | 1'18"   2'05" |
CORRETT.TOT.| 1'10"     16'58" | 6'52   54'55" | 1'50"   3'05" |
============+==================+===============+===============+
```

Figura 10.12 Valutazione sperimentale dell'efficienza della verifica